Introducing PARALLEL: Stata Module for Parallel Computing [draft]

George Vega Yon[*]
Research Department
Chilean Pension Supervisor

This version September 26, 2012

Abstract

Inspired in the R library "snow" and to be used in multicore CPUs, `parallel` implements parallel computing methods through an OS's shell scripting (using Stata in batch mode) to accelerate computations. By splitting the dataset into a determined number of clusters this module repeats a task simultaneously over the data-clusters, allowing to increase efficiency between two and five times, or more depending on the number of cores of the CPU. Without the need of StataMP, `parallel` is, to the author's knowledge, the first user contributed Stata module to implement parallel computing.

Keywords: Parallel Computing, High Performance Computing, Simulation Methods, CPU

JEL Codes: C87, C15, C61

[*]gvega at spensiones.cl. Thanks to Damian C. Clarke, Félix Villatoro and Eduardo Fajnzylber and the Research team of the Chilean Pension Supervisor for their valuable contributions. The usual disclaimers applies.

1 Introduction

Currently home computers are arriving with extremely high computational capabilities. Multicore CPUs, standard in today's industry, expand the boundaries of productivity for the so called multitasking-users. Motivated by the video games industry, manufacturers have forged a market for low-cost processing units with the number-crunching horsepower comparable to that of a small supercomputer (?, ?).

In the same way, data availability has improved in a significant manner. Big-Data is an active topic by computer scientists and policy-makers considering the number of open-data initiatives taking place around the globe giving access to administrative data. Resources that, despite being available to researchers and policy makers, have not been exploited as they should.

The limited use of these social data resources is not a coincidence. As Gary King states in ? (?), issues involving privacy, standardized management and lack of statistical computing tools are still unsolved both for social scientists, and policy-makers. `parallel` aims to make a contribution to these issues.

2 Parallel Computing

In simple terms, parallel computing is the simultaneous use of multiple compute resources to solve a computational problem (?, ?). Parallelism can take place through several levels starting from (a) bit level, (b) instruction level, (c) data level and up to (d) task level. `parallel` uses data level parallelism, which basically consists in simultaneously repeating an instruction over independent groups of data.

Using parallel computing allows the user to drastically decrease the time required to complete a computational problem. This is especially important for data scientists such as empirical economists and econometricians, given that in this way it is possible to implement algorithms characterized by a large number of calculations or, in the case of Stata, code interpretation such as flow-control statements like loops.

Flynn's taxonomy classification provides a simple way to classify the type of parallelisms that researchers may require. Based on the type of instruction and the type of data, both of which can be single or multiple, Flynn identifies four classes of computer architectures (?, ?).

According to this classification, `parallel` would follow a Single Instruction

Table 1: Flynn's taxonomy

	Single Instruction	Multiple Instruction
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD

Multiple Data (SIMD) design, where the parallelism takes place by repeating a single task over groups of data. Even though this is a very simple form of parallelism, significant improvements can be achieved from its use.

With regards to the size of the improvements, following Amdahl's law, the speed improvement S depends upon (a) the proportion of potential deserialization P; this is, the proportion of the algorithm that can be implemented in a parallel fashion, and (b) the number of fractions to be split, N.

$$S = \frac{1}{(1-P) + \frac{P}{N}} \tag{1}$$

Thus, if P tends to one, the potential speed gains S will be equal to N. In the same way, as P approaches to zero, any parallelization attempt will be worthless[1]. Considering this, theoretically, SIMD class may reach perfect scaling.

3 Parallel Computing in Econometrics

Even though parallel computing is not a new idea[2], economics has drawn relatively little from parallel computing. In ? (?) efforts are made by introducing a new C library based on matrix programming language Ox with the objective to promote parallel computing in social sciences. Using alternative approach, and after experimenting with GPU based parallel computing, ? (?) show that using this approach to solve a Real Business Cycle model and using a consumer level NVIDIA video card, it is possible to reach speed improvements of around two hundred times, with this being a lower bound.

Statistical packages are also advancing in this line. Matlab provides its own Parallel Computing Toolbox[3] which makes it possible to implement parallel computing methods through multicore computer, GPUs and computer clusters.

[1] A good technical review on modern themes on parallel computing is provided by ? (?)

[2] Applied sciences, physics, computer science and industry have taken considerable advantage of it.

[3] http://www.mathworks.com/products/parallel-computing/

GNU open-source R also has several libraries to implement parallel computing algorithms such as parallel, snow, and so forth[4]. And Stata with its Multi-Processor edition, StataMP, implementing bit level parallelization makes it possible to achieve up to (or greater than) constant scale speed improvements (?, ?).

4 PARALLEL: Stata module for parallel computing

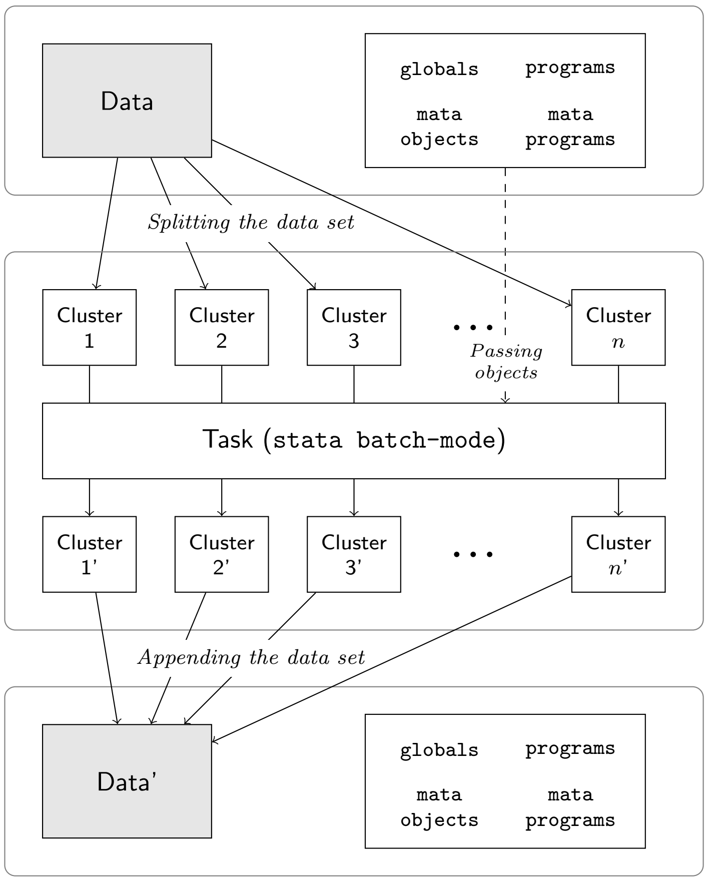
Inspired by the R library "snow" and to be used in multicore CPUs, `parallel` implements parallel computing methods through OS's shell scripting (using Stata in batch mode) to speedup computations by splitting the dataset into a determined number of clusters[5] in such a way to implement a data parallelism algorithm.

As exposed in Figure 1, right after `parallel` splits the dataset into n clusters it starts n new independent stata instances in batch mode over which the same task is simultaneously executed. By default all the loaded instance globals and, optionally, programs and mata objects/programs are passed through. After every cluster stops the resulting datasets are appended and returned to the current stata instance without modifying other elements.

[4]For more details see CRAN Task View on High-Performance and Parallel Computing With R `http://cran.r-project.org/web/views/HighPerformanceComputing.html`

[5]It is important to distinguish between two different ways to understand a cluster. In computer science a *cluster*, or *computer cluster*, refers to a set of computers connected so that they to work as a single system. Here, in the other hand, as this module is intended to be use by statisticians and social scientist in general, I refer to a cluster as a package of data which in this case does not necessary contains related observations (clustered).

Figure 1: How `parallel` works



Starting (current) `stata` instance loaded with data plus user defined `globals`, `programs`, `mata objects` and `mata programs`

A new `stata` instance (batch-mode) for every data-clusters. Programs, globals and mata objects/programs are passed to them.

The same algorithm (task) is simultaneously applied over the data-clusters.

After every instance stops, the data-clusters are appended into one.

Ending (resulting) `stata` instance loaded with the new data.

User defined `globals`, `programs`, `mata objects` and `mata programs` remind unchanged.

The number of efficient computing clusters depends upon the number of physical cores (CPUs) with which your computer is built, e.g. if you have a quad-core computer, the correct cluster setting should be four. In the case of simultaneous multithreading, such as that from Intel's hyper-threading technology (HTT), setting `parallel` following the number of processors threads, as it was expected, hardly results into a perfect speedup scaling. In spite of it, after several tests on HTT capable architectures, the results of implementing `parallel` according to the machines physical cores versus its logicals shows small though significant differences.

`parallel` is especially handy when it comes to implementing loop-based simulation models (or simply loops), Stata commands such as reshape, or any job that (1) can be repeated through data-blocks, and (2) routines that processes big datasets.

In the case of (pseudo) random number generation, `parallel` allows to set one seed per cluster with the option `seeds(`*numlist*`)`.

At this time `parallel` has been successfully tested in Windows and Unix machines. Tests using Mac OS are still pending.

4.1 Syntax

`parallel`'s simplistic syntax is one of its key features. In order to use it, only two commands needed be entered: `parallel setclusters`, which tells `parallel` how many blocks of data are to be built (and thus how many Stata batch mode instances are needed to be run), and `parallel do` (to run a do-file in parallel) or `parallel:` (to use it as a prefix to parallelize a command). More, detailed, as specified in its help-file:

Setting the number of clusters (blocks of data)

```
parallel setclusters #
```

Parallelizing a dofile

```
parallel do filename [,options]
```

Parallelizing a stata command

```
parallel [, options]:  stata_cmd
```

Removing auxiliary files

```
parallel clean [parallelid]
```

5 Results

In what follows, I present some results testing the efficiency of `parallel` over different hardware and software configurations using StataSE. The tables summarize the results. The first row presents the time required to complete the task using a single processor (cluster or thread as a computer scientist may prefer). The second line shows the total time spent to complete the task using `parallel`, while the following three distinguish between setup (time taken to prepare the algorithm), compute (execution time of the task as such) and finish (mainly appending the datasets). Finally the last two show the ratio of CPU time over compute and total time respectively. Every time measure is presented in seconds.

It is important to consider that both setup time and finishing time increases as the problem size scales up.

All the test were performed checking whether if the parallel implementation returned different results from the serial implementation.

5.1 Serial replacing using a loop

This first test consists on, after a generation of N pseudo-random values, using stata's `rnormal()` function, replacing each and every one of the observations in a serial way (loop) starting from 1 to N. The observation's variable was replaced using the PDF of the normal distribution.

$$f(x) = \frac{1}{\sqrt{2\pi}} e^{\frac{-x^2}{2}} \tag{2}$$

The code to be parallelized is

```
--------------- begin of do-file -------------
local size = _N

forval i=1/‘size’ {
   qui replace x = 1/sqrt(2*‘c(pi)’)*exp(-(x^2/2)) in ‘i’
}
--------------- end of do-file ---------------
```

which is contained inside a do-file named "myloop.do", and can be executed through four clusters with `parallel` as it follows

```
parallel setclusters 4

parallel do myloop.do
```

This algorithm was repeated over

$$N \in \{10,000; 100,000; 1,000,000; 10,000,000\}$$

Table 2: Serial replacing using a loop on a Windows Machine (2 clusters)

	Problem Size			
	10.000	100.000	1.000.000	10.000.000
CPU	0.12	1.22	11.72	58.03
Total	0.81	1.53	7.96	36.13
Setup	0.02	0.08	0.56	2.86
Compute	0.55	1.20	7.10	32.65
Finish	0.25	0.25	0.30	0.62
Ratio (compute)	0.23	1.01	1.65	1.78
Ratio (total)	0.15	0.80	1.47	1.61

Tested on an Intel Core i5 M560 (dual-core) machine

Table 3: Serial replacing using a loop on a Linux Server (4 clusters)

	Problem Size			
	10.000	100.000	1.000.000	10.000.000
CPU	0.25	1.79	17.64	176.16
Total	0.45	1.00	5.14	42.61
Setup	0.02	0.11	0.81	5.16
Compute	0.23	0.67	3.86	35.98
Finish	0.21	0.23	0.46	1.47
Ratio (compute)	1.13	2.68	4.57	4.90
Ratio (total)	0.56	1.79	3.43	4.13

Tested on an Intel Xeon X470 (octa-core) machine

5.2 Reshaping a large database

Reshaping a large database is always a time consuming task. This example shows the execution of Stata's `reshape wide` command for a large administrative dataset which contains unemployment insurance solicitations made by its users, i.e. a panel dataset.

The task was repeated over different sample sizes

$$N \in \{1,000; 10,000; 100,000; 1,000,000; 10,000,000\}$$

Using `parallel` prefix syntax, the command is as it follows

```
parallel, by(numcue) force:reshape wide ///
    tipsolic rutemp opta derecho ngiros, i(numcue) j(tiempo)
```

where the options `by` consider observations grouping in so as not to split in between and `force`, jointly used with `by`, tells `parallel` not to check whether if the dataset is actually sorted by, in this case, `numcue`.

Table 4: Reshaping wide a large database on a Windows Machine (2 clusters)

	Problem Size		
	100000	1000000	2000000
CPU	4.95	48.03	101.38
Total	5.04	39.58	81.40
Setup	0.23	1.15	1.92
Compute	4.34	34.96	72.14
Finish	0.47	3.46	7.35
Ratio (compute)	1.14	1.37	1.41
Ratio (total)	0.98	1.21	1.25

Tested on an Intel Core i5 M560 (dual-core) machine

Table 5: Reshaping wide a large database on a Windows Machine (4 clusters with HTT)

	Problem Size		
	100000	1000000	2000000
CPU	4.91	50.48	106.03
Total	4.42	29.00	69.69
Setup	0.45	2.06	3.51
Compute	3.84	23.57	55.86
Finish	0.13	3.37	10.31
Ratio (compute)	1.28	2.14	1.90
Ratio (total)	1.11	1.74	1.52

Tested on an Intel Core i5 M560 (dual-core) machine

Table 6: Reshaping wide a large database on a Linux Server (4 clusters)

	Problem Size		
	100.000	1.000.000	5.000.000
CPU	9.00	101.94	564.13
Total	4.41	43.92	317.60
Setup	0.77	1.43	10.03
Compute	3.17	38.52	283.05
Finish	0.47	3.98	24.53
Ratio (compute)	2.84	2.65	1.99
Ratio (total)	2.04	2.32	1.78

Tested on an Intel Xeon X470 (octa-core) machine

Table 7: Reshaping wide a large database on a Linux Server (8 clusters)

	Problem Size		
	100000	1000000	5000000
CPU	9.21	98.87	534.34
Total	3.95	47.08	233.90
Setup	0.94	2.63	18.37
Compute	2.46	40.13	188.47
Finish	0.55	4.32	27.07
Ratio (compute)	3.74	2.46	2.84
Ratio (total)	2.33	2.10	2.28

Tested on an Intel Xeon X470 (octa-core) machine

5.3 Extended test: Monte Carlo Simulation

In ? (?) a simple Monte Carlo experiment is perform which simulates the performance of a estimator of sample mean, $\bar{x}$, in a context of heteroskedasticity.

The model to be

$$y_i = \mu + \epsilon_i \sim N(0, \sigma^2) \tag{3}$$

Let ϵ be a $N(0,1)$ variable multiplied by a factor cz_i, where z_i varies over i. We will vary parameter c between 0.1 and 1.0 and determine its effect on the point and interval estimates of μ; as a comparison, we will compute a second random variable which is homoskedastic, with the scale factor equalling $c\bar{z}$.

From web dataset `census2` the variables `age`, `region` (which can be 1, 2, 3 or 4) and the mean of `region` are used as μ, z_i and $\bar{z}$ respectively.

The simulation program, stored in “mcsimul1.ado”, is defined as

```
--------------- begin of ado-file ------------
program define mcsimul1, rclass
    version 10.0
    syntax [, c(real 1)]

    tempvar e1 e2
    gen double ‘e1’=invnorm(uniform())*‘c’*zmu
    gen double ‘e2’=invnorm(uniform())*‘c’*z_factor

    replace y1 = true_y + ‘e1’
    replace y2 = true_y + ‘e2’

    summ y1
    return scalar mu1 = r(mean)
    return scalar se_mu1 = r(sd)/sqrt(r(N))

    summ y2
    return scalar mu2 = r(mean)
    return scalar se_mu2 = r(sd)/sqrt(r(N))

    return scalar c = ‘c’
end
--------------- end of ado-file --------------
```

In what is next, the do-file that runs the simulation, stored as “montecarlo.do”, it is compound of two parts: (a) setting the iteration range by which c is going to vary, and (b) looping over the selected range. For the first part the do-file uses the local macro `pll_instance` which is the numer of the parallel stata instance running, thus there are as many as clusters have been declared, number available with the global macro PLL_CLUSTERS. This way, if the macro PLL_CLUSTERS equals two and the macro `pll_instance` equals une, then the range will be defined from one to five[6].

```
--------------- begin of do-file -------------
// Defining the loop range
local num_of_intervals = 10
if length("‘pll_id’") == 0 {
    local start = 1
    local end = ‘num_of_intervals’
}
else {
    local ntot = floor(‘num_of_intervals’/$PLL_CLUSTERS)
    local start = (‘pll_instance’ - 1)*‘ntot’ + 1
    local end = (‘pll_instance’)*‘ntot’
    if ‘pll_instance’ == $PLL_CLUSTERS local end = 10
}

local reps 1000

// Loop
forval i=‘start’/‘end’ {
    qui webuse census2, clear
    gen true_y = age
    gen z_factor = region
    sum z_factor, meanonly
    scalar zmu = r(mean)
    qui {
        gen y1 = .
        gen y2 = .
        local c = ‘i’/10
        set seed ‘c’
        simulate c=r(c) mu1=r(mu1) se_mu1 = r(se_mu1) ///
                mu2=r(mu2) se_mu2 = r(se_mu2), ///
                saving(cc‘i’, replace) nodots reps(‘reps’): ///
                mcsimul1, c(‘c’)
    }
}
--------------- end of do-file ---------------
```

This do-file will be executed from stata using `parallel do` syntax. As there

[6]Note that if the local macro `pll_id`, which contains a special random number that identifies an specific `parallel` run (more details in its help file), length is zero it means that the do-file is not running in parallel mode, thus it is been executed in a serial way where the loop range starts from one to ten.

is no need of splitting any dataset (these are loaded every time that the main `loop_simul.do`'s loop runs), we add the option `nodata`. This way the main do-file will look like this Finally, the do-file from which parallel runs the simulation

```
--------------- begin of do-file -------------
clear all
parallel setclusters 5
parallel do loop_simul.do, nodata
--------------- end of do-file ---------------
```

By this `parallel` will start, in this case, five new independent stata instances, each one looping over the ranges 1/2, 3/4, 5/6, 7/8 and 9/10 respectively.

Most of the code has been exactly copied with the exception of the addition of a new code line `set seed`. In order to be able to compare both serial and parallel implementations of the algorithm it was necessary to set a particular seed for each loop, inside "montecarlo.do" right before `simulate` command.

Table 8: Monte Carlo Experiment on a Windows Machine

	Number of Clusters	
	2	4
CPU	28.50	28.92
Total	17.49	18.30
Setup	0.11	0.12
Compute	17.27	18.07
Finish	0.11	0.11
Ratio (compute)	1.65	1.60
Ratio (total)	1.63	1.58

Tested on a Intel Core i5 M560 (dual-core) machine

Table 9: Monte Carlo Experiment on a Linux Server

	Number of Clusters		
	2	3	5
CPU	40.97	39.01	36.44
Total	18.35	15.20	7.62
Setup	0.11	0.11	0.12
Compute	18.13	14.99	7.41
Finish	0.10	0.10	0.10
Ratio (compute)	2.26	2.60	4.92
Ratio (total)	2.23	2.57	4.78

Tested on a Intel Xeon X470 (octa-core) machine

6 Concluding Remarks

As stated by computer scientist Ph.D. Wen-Mei Hwu[7], parallel computing is the future of supercomputing, and giving the computer industry's fast pace of development, scientists in various areas are making real efforts to promote its usage. In spite of this, many of social scientist do not work with it due to its lack of user-friendly implementations.

In the case of Stata, `parallel` is, to the authors knowledge, the first public user-contribution to parallel computing. The major benefits, measured in terms of speedups, can be reached by simulation models and non-vectorized operations such as control-flow statements. Speed gains are directly related to the proportion of the algorithm that can be de-serialized and the number of processors with which the parallelization is made, making possible reach near to constant scale improvements.

Notwithstanding the simplicity of this type of parallelism, giving to its easy way, it seems like a worthy activity for a wide range of social scientists and researchers.

[7] Chief Scientist at *Parallel Computing Institute* `http://parallel.illinois.edu/news-and-media/gpus-path-future`